Unranking Algorithms for Combinatorial Structures

X. Molinero and J. Vives

***Abstract*—We present an implementation of some unlabeled and labeled unranking algorithms for the open-source algebraic combinatorics package MUPAD-COMBINAT of the computer algebra system MUPAD. We have compared our implementation with the previous versions. All our algorithms improve the previous ones with respect to the required CPU time. Moreover, we have also developed unranking algorithms applied to some unlabeled and labeled admissible operators that are not still implemented in the package MUPAD-COMBINAT. These algorithms are also able to develop some combinatorial structures useful to generate molecules applied to chemistry and influence graphs applied to game theory and social networks, among other topics.**

***Index Terms*—Unranking Algorithms for Combinatorial Structures, package MUPAD-COMBINAT, Generating Molecules, Generating Influence Games.**

I. INTRODUCTION

The problem of *unranking* asks for the generation of the ith combinatorial object of size n in some combinatorial class $\mathcal{A}$, according to some well defined order among the objects of size n of the class. Efficient unranking algorithms have been devised for many different combinatorial classes, like binary and Cayley trees, Dyck paths, permutations, strings or integer partitions, but most of the work in this area concentrates in efficient algorithms for particular classes, whereas we aim at generic algorithms that apply to a broad family of combinatorial classes. The problem of unranking is intimately related with its converse, the *ranking* problem, as well as with the problems of random generation and exhaustive generation of all combinatorial objects of a given size [23], [12].

The *ranking* problem is the following: Given a combinatorial class and an object from that class, compute the *rank* of the given object, according to some previously fixed order.

The *random generation problem* consists in generating a combinatorial object of a given combinatorial class and size uniformly at random.

The *unranking* problem generates a combinatorial object whose rank and size are given, according to some fixed order.

The *iteration or exhaustive generation* gives all objects of a given combinatorial class and size, according to some previously fixed order.

The interest of this whole subject is witnessed by the vast number of research papers and books that has appeared in over five decades (see, for instance, [34], [19], [15], [14], [18], [35], [17], [27], [26], [28], [6]).

X. Molinero is with the Department of Applied Mathematics III, Universitat Politècnica de Catalunya, E-08240 Manresa, SPAIN. E-mail: xavier.molinero@upc.edu. X. Molinero was partially funded by grant MTM2012-34426/FEDER of the "Spanish Economy and Competitiveness Ministry".

J. Vives is with the Department of Design and Programming of Electronic Systems, Universitat Politècnica de Catalunya, E-08240 Manresa, SPAIN. E-mail: jvives@epsem.upc.edu.

[23], [21], [20] designed *generic* unranking algorithms for a large family of (unlabeled and labeled) combinatorial classes, namely, those which can be inductively built from the basic ϵ-class (a class which contains only one object of size 0), atomic classes (classes that contain only one object of size 1 or *atom*) and a collection of admissible combinatorial operators: disjoint unions, products, sequence, set, cycles, etc. Now we use such techniques to implement those algorithms in MUPAD [4], [25]. In the open-source algebraic combinatorics package MUPAD-COMBINAT [2] for the computer algebra system MUPAD there are implemented the unranking for some admissible combinatorial operators. First, we have improved such implementation for unlabeled unions and products (and sequences). Afterwards, we have also implemented other operators as unlabeled sets and powersets (with and without restrictions), and labeled unions, products, sequences and sets (with and without restrictions if it is possible) and cycles.

The paper is organized as follows. In Section II we briefly review basic definitions and concepts, the unranking algorithms and the theoretical analysis of their performance. In Section III, we compare the required CPU time of our implementation with the required CPU time of the current implementation in the package MUPAD-COMBINAT of the computer algebra system MUPAD. Section IV establishes our conclusions and future work.

Finally, Appendix presents the specification of our implementation and sketches some details about how it works.

II. PRELIMINARIES

The goal of this paper is to generate efficient unranking algorithms of combinatorial structures in the computer algebra system MUPAD. All these algorithms require to count how many objects of each size there are, that is, given a specification of a class and a size, we need to compute the number of objects with the given size. Hence, we will only deal with (some of) the so-called *admissible combinatorial classes* [10], [11]. Those are constructed from *admissible operators*, operations over classes that yield new classes, such that the number of objects of a given size in the new class can be computed from the number of objects of the same or smaller sizes in the constituent (sub)classes. In this paper we essentially just consider unlabeled objects (those whose atoms are indistinguishable[1]) built from these admissible combinatorial operators. However, similar algorithms have also been implemented for labeled objects.

For both labeled and unlabeled classes, the *finite* specifications are generated from the ϵ-class, atomic classes (undistinguishable for unlabeled objects and distinguishable for labeled

[1]On the contrary, each of the n atoms of a *labeled* object of size n bears a distinct label drawn from the numbers 1 to the size of the given object.

Unlabeled class	Specification
Binary trees	$\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}$
Unary-binary trees or	
Motzkin trees	$\mathcal{M} = Z + Z \times \mathcal{M} + Z \times \mathcal{M} \times \mathcal{M}$
Integer partitions	$\mathcal{P} = \mathsf{Set}(\mathsf{Seq}(Z, \text{card} \geq 1))$
Integer compositions	$\mathcal{C} = \mathsf{Seq}(\mathsf{Set}(Z, \text{card} \geq 1))$
Non-ordered rooted trees or	
Rooted unlabeled trees	$\mathcal{T} = Z \times \mathsf{Set}(\mathcal{T})$
Binary sequences	$\mathcal{A} = \mathsf{Seq}(Z + Z)$
Non plane ternary trees	$\mathcal{D} = Z + \mathsf{Set}(\mathcal{D}, \text{card}{=}3)$
Integer partitions with	
distinct parts	$\mathcal{E} = \mathsf{PowerSet}(\mathsf{Seq}(Z, \text{card}{\geq}1))$
Necklaces	$\mathcal{B} = \mathsf{Cycle}(\mathsf{Set}(Z, \text{card}{\geq}1))$

Fig. 1. Examples of unlabeled classes and their specifications

Labeled class	Specification
Binary plane trees	$\mathcal{B} = Z + \mathcal{B} \star \mathcal{B}$
General plane trees	$\mathcal{G} = Z \star \mathsf{Seq}(\mathcal{G})$
Set partitions	$\mathcal{P} = \mathsf{Set}(\mathsf{Set}(Z, \text{card} \geq 1))$
Non-plane binary trees	$\mathcal{D} = Z + \mathsf{Set}(\mathcal{D}, \text{card} = 2)$
Non-plane ternary trees	$\mathcal{E} = Z + \mathsf{Set}(\mathcal{E}, \text{card} = 3)$
Hierarchies	$\mathcal{J} = Z + \mathsf{Set}(\mathcal{J}, \text{card} \geq 2)$
3-balanced hierarchies	$\mathcal{I} = \mathsf{Set}(\mathsf{Set}(\mathsf{Set}(Z, \text{card} \geq 1), \text{card} \geq 1))$
Surjections	$\mathcal{V} = \mathsf{Seq}(\mathsf{Set}(Z, \text{card} \geq 1))$
Cayley (Non-plane) trees	$\mathcal{T} = Z \star \mathsf{Set}(\mathcal{T})$
Functional graphs	$\mathcal{F} = \mathsf{Set}(\mathsf{Cycle}(\mathcal{T}))$

Fig. 2. Examples of labeled classes and their specifications

objects), and combinatorial operators including disjoint union ('+'), Cartesian unlabeled product ('$\times$'), labeled product ('$\star$'), sequence ('Seq'), powerset ('PowerSet'), set ('Set')[2], cycle ('Cycle') and substitution ('Subst'), and sequence, powerset, set and cycle with restricted cardinality. Figures 1 and 2 gives a few examples of unlabeled and labeled admissible classes, respectively.

For the rest of this paper, we will use calligraphic uppercase letters to denote classes: $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... Given a class $\mathcal{A}$ and a size n, $\mathcal{A}_n$ will denote the subset of objects of size n in $\mathcal{A}$.

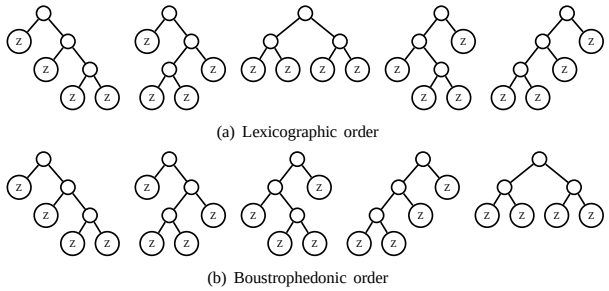
The order $\prec_{\mathcal{C}_n}$ among the objects of size n for a class $\mathcal{C} = \mathcal{A} + \mathcal{B}$ is naturally defined by $\gamma \prec_{\mathcal{C}_n} \gamma'$ if both γ and γ' belong to the same class (either $\mathcal{A}_n$ or $\mathcal{B}_n$) and $\gamma \prec \gamma'$ within their class, or if $\gamma \in \mathcal{A}_n$ and $\gamma' \in \mathcal{B}_n$. It is then clear that although $\mathcal{A} + \mathcal{B}$ and $\mathcal{B} + \mathcal{A}$ are isomorphic ("the same class"), these two specifications induce quite different orders. The unranking algorithm for disjoint unions compares the given rank with the cardinality of $\mathcal{A}_n$ to decide if the sought object belongs to $\mathcal{A}$ or to $\mathcal{B}$ and then solves the problem by recursively calling the unranking on whatever class ($\mathcal{A}$ or $\mathcal{B}$) is appropriate.

For Cartesian products the order in $\mathcal{C}_n = (\mathcal{A} \times \mathcal{B})_n$ depends on whether $\gamma = (\alpha, \beta)$ and $\gamma' = (\alpha', \beta')$ have first components of the same size. If $|\alpha| = |\alpha'| = j$ then we have $\gamma \prec_{\mathcal{C}_n} \gamma'$ if $\alpha \prec_{\mathcal{A}_j} \alpha'$ or $\alpha = \alpha'$ and $\beta \prec_{\mathcal{B}_{n-j}} \beta'$. But when $|\alpha| \neq |\alpha'|$, we must provide a criterion to order γ and γ'. The *lexicographic* order stems from the specification

$$\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \ldots + \mathcal{A}_n \times \mathcal{B}_0,$$

in other words, the smaller object is that with smaller first component. On the other hand, the *boustrophedonic* order is

[2]Also denoted by 'multisets' (MultiSet) to emphasize that repetition is allowed.



(a) Lexicographic order

(b) Boustrophedonic order

Fig. 2. Binary trees of size 4.

induced by the specification

$$\mathcal{C}_n = \mathcal{A}_0 \times \mathcal{B}_n + \mathcal{A}_n \times \mathcal{B}_0 + \mathcal{A}_1 \times \mathcal{B}_{n-1} + \mathcal{A}_{n-1} \times \mathcal{B}_1 + \mathcal{A}_2 \times \mathcal{B}_{n-2} + \mathcal{A}_{n-2} \times \mathcal{B}_2 + \ldots,$$

in other words, we consider that the smaller pairs of total size n are those whose $\mathcal{A}$-component has size 0, then those with $\mathcal{A}$-component of size n, then those with $\mathcal{A}$-component of size 1, and so on. Figure II shows the lists of unlabeled binary trees of size 4 in lexicographic (a) and boustrophedonic order (b).

Of course, other orders are also possible, but they either do not help improving the performance of unranking or they are too complex to be useful or applied.

For sequences we use the natural isomorphic from unlabeled and labeled union and product:

$$\mathsf{Seq}(\mathcal{A}) = \epsilon \ + \mathcal{A} \times \mathsf{Seq}(\mathcal{A})$$

and

$$\mathsf{Seq}(\mathcal{A}) = \epsilon \ + \mathcal{A} \star \mathsf{Seq}(\mathcal{A}),$$

respectively.

On the other hand, for unlabeled powersets, among some natural orders (see [21], [23]) we can choose

$$\mathsf{PowerSet}(\mathcal{A}) = \epsilon + \bigcup_{n>0} \bigcup_{j=1}^{n} \bigcup_{k=n \div j}^{1} \Big(\mathsf{PowerSet}(\mathcal{A}_j, \text{card}{=}\ k) \times \mathsf{PowerSet}_{n-kj}(\mathcal{A}_{>j})\Big)$$

where

$$\mathsf{PowerSet}(\mathcal{A}_j, \text{card}{=}\ k) = \bigcup_{\alpha \in \mathcal{A}_j} \Big(\alpha \times \mathsf{PowerSet}(\mathcal{A}_j^{(\succ \alpha)}, \text{card}{=}\ k-1)\Big),$$

being $\mathcal{A}^{(\succ \alpha)} = \{\alpha' \in \mathcal{A} : \alpha' \succ \alpha\}$, and $\mathsf{PowerSet}(\mathcal{A}_{>j})$ is a powerset with $\mathcal{A}$-components of size at least equal to $j+1$. Other orders described in [21], [23] do not change the complexity and they could also be easily adapted to our implementation.

For unlabeled sets we have analogous isomorphisms but allowing repetitions.

On the other hand, for labeled sets and cycles see the isomorphisms described with the so-called *boxed product* in [23].

The theoretical performance of these unranking algorithms is summarized in [23], [20].

Theorem 1: The worst-case time complexity of unranking for objects of size n in any admissible unlabeled or labeled class $\mathcal{A}$ using lexicographic ordering is of $\mathcal{O}(n^2)$ arithmetic operations.

Theorem 2: The worst-case time complexity of unranking for objects of size n in any admissible unlabeled or labeled class $\mathcal{A}$ using boustrophedonic ordering is of $\mathcal{O}(n \log n)$ arithmetic operations.

III. Our implementation v.s. MuPAD-Combinat implementation

In this section we compare our implementation[3] for unranking in MuPAD with the current implementation of the package MuPAD-Combinat(using MuPAD Pro 4.0). All our experiments run under Linux in a AMD64X2 4400 at 2.2 GHz with 4 Gb of RAM, and they use the basic facilities for counting already provided by the package MuPAD-Combinat.

For instance, the interface for binary trees has the following inputs:

```
spec := {B = Union(Z, Prod(B, B)),};
p1 := combinat::
      decomposableObjects(spec,
                          Lexi/Bous);
p1::unrank(rank, size);
```

where `spec` is the specification[4], `Lexi` or `Bous` forces the lexicographic or boustrophedonic order, respectively, and `rank` and `size` are the considered rank and size, respectively. Thus, the following commands provide all binary trees of size 8 in lexicographic order:

```
spec := {B = Union(Z, Prod(B, B))};
p1 := combinat::
      decomposableObjects(spec, Lexi);
for i from 0 to p1::count(8) - 1 do
  p1::unrank(i, 8);
end_for
```

Notice that, in general, `p1::count(size)` returns the number of objects of `p1` with size `size`.

The selected collection for our experiments are two unlabeled classical classes: binary trees ($\mathcal{B} = Z + \mathcal{B} \times \mathcal{B}$) and, unary-binary trees or Motzkin trees ($\mathcal{M} = Z + Z \times \mathcal{M} + Z \times \mathcal{M} \times \mathcal{M}$).

Essentially, we have used two techniques in our implementation. First, we have appropriately used the command `option remember`. Second, we have also used some pre–computed tables (using dichotomic search among other techniques) to store the counting of each class and size considered. Moreover, the access to the indices of such tables is substantially faster than the access to the indices in the previous version established in the command `count` of MuPAD. Note that to count the number of objects of a given size and class we have used ordinary and exponential generating functions (o.g.f. and e.g.f) for unlabeled and labeled classes with the relation given in Tables I and II, respectively, where the coeficient of z^n gives the number of objects of size n in the corresponding class.

[3]It is available on request from the first author.

[4]The first class defined in the specification is the considered class (`B` in this case).

Class	Ordinary Gen. Func.
ϵ (empty class)	1
Z (Atomic class)	Z
$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$
$\mathcal{A} \times \mathcal{B}$	$A(z) \cdot B(z)$
$\mathsf{Seq}(\mathcal{A})$	$\frac{1}{1-A(z)}$
$\mathsf{PowerSet}(\mathcal{A})$	$\exp\left(\sum_{n>0} (-1)^{n-1} \frac{A(z^n)}{n}\right)$
$\mathsf{Set}(\mathcal{A})$	$\exp\left(\sum_{n>0} \frac{A(z^n)}{n}\right)$
$\mathsf{Cycle}(\mathcal{A})$	$\sum_{n>0} \frac{\phi(n)}{n} \log\left(\frac{1}{1-A(z^n)}\right)$
$\mathsf{Subst}(\mathcal{A}, \mathcal{B})$	$A(B(z))$

TABLE I
Relation of ordinary generating functions for counting the number of objects of a specific size and an unlabeled admissible combinatorial class.

Class	Exponential Gen. Func.
ϵ (empty class)	1
Z_i (Atomic class)	Z
$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$
$\mathcal{A} \star \mathcal{B}$	$A(z) \cdot B(z)$
$\mathsf{Seq}(\mathcal{A})$	$\frac{1}{1-A(z)}$
$\mathsf{Set}(\mathcal{A})$	$\exp(A(z))$
$\mathsf{Cycle}(\mathcal{A})$	$\log\left(\frac{1}{1-A(z)}\right)$
$\mathsf{Subst}(\mathcal{A}, \mathcal{B})$	$A(B(z))$

TABLE II
Relation of exponential generating functions for counting the number of objects of a specific size and a labeled admissible combinatorial class.

Next, Tables III and IV show the improvement of the average CPU time (in milliseconds) for unlabeled binary trees and for unlabeled Motzkin trees. We have pre–computed the counting tables and, afterwards, we have generated 10000 random objects of the considered class and size. τ_T is our average time required to unrank a random rank of the considered class and size, τ'_T is MuPAD-Combinat average time required to unrank a random rank of the considered class and size, and ρ is the ratio τ_T/τ'_T. For any case, it looks as the improvements tend to be stable when the size n increase. For lexicographic binary trees it approaches to $\rho = 0.33$, for Boustrophedonic binary trees it approaches to $\rho = 0.40$, for lexicographic Motzkin trees it approaches to $\rho = 0.28$, and for Boustrophedonic Motzkin trees it approaches to $\rho = 0.41$. Thus, all results are satisfactorily better. Note that even the pre–computed tables require some CPU time, the average CPU time (when the number of generated objects increase) of our implementation substantially improves the previous one. We have meaningfully improved the average CPU time required to generate a random unranking: In lexicographic order, our implementation spends about 30% of the CPU time of the

Lexicographic order				Boustrophedonic order			
Size	τ_T	τ'_T	ρ	Size	τ_T	τ'_T	ρ
25	1.24	4.55	0.27	25	0.90	2.95	0.30
50	3.63	11.95	0.30	50	2.50	6.52	0.38
75	6.08	20.17	0.30	75	4.27	10.24	0.41
100	9.27	29.65	0.31	100	6.20	13.97	0.44
125	12.46	39.98	0.31	125	6.99	17.91	0.39
150	16.54	51.73	0.31	150	8.39	20.80	0.40
175	21.37	66.16	0.32	175	9.67	24.90	0.38
200	26.85	79.93	0.33	200	11.63	28.87	0.40

TABLE III
AVERAGE CPU TIME (IN MILLISECONDS) FOR BINARY TREES, $\mathcal{B}$=Z+$\mathcal{B}$×$\mathcal{B}$. THE CPU TIME REQUIRED TO CALCULATE THE PRE–COMPUTED TABLES IN OUR UNRANKING IS 380 MILLISECONDS.

Lexicographic order				Boustrophedonic order			
Size	τ_T	τ'_T	ρ	Size	τ_T	τ'_T	ρ
25	1.12	3.24	0.34	25	1.06	2.57	0.41
50	2.60	9.33	0.27	50	2.71	5.75	0.47
75	4.81	16.61	0.28	75	4.19	9.74	0.43
100	7.18	25.72	0.27	100	5.95	13.76	0.43
125	9.65	34.27	0.28	125	8.01	17.66	0.45
150	12.28	42.57	0.28	150	9.80	22.43	0.43
175	15.42	54.44	0.28	175	11.49	27.76	0.41
200	18.83	67.08	0.28	200	13.56	32.67	0.41

TABLE IV
AVERAGE CPU TIME (IN MILLISECONDS) FOR MOTZKIN TREES, $\mathcal{M}$=Z+Z×$\mathcal{M}$+Z×$\mathcal{M}$×$\mathcal{M}$. THE CPU TIME REQUIRED TO CALCULATE THE PRE–COMPUTED TABLES IN OUR UNRANKING IS 690 MILLISECONDS.

previous version; and, in boustrophedonic order, it spends about 40% of the CPU time of the previous version.

Sequences are done from unions and products, thus the timing improvements have similar advantages.

On the other hand, we have also done some experiments with classes that involve sets or cycles, for instance, we have considered the so-called *functional graphs* defined by $\mathcal{F}$ = Set(Cycle($\mathcal{T}$)) with $\mathcal{T}$ = Z×Set($\mathcal{T}$)). In such cases, our implementation takes similar CPU time than the previous examples, except for the pre–computed tables about the counting of the number of object for each size. In general, it takes more CPU time when classes are described from sets and powersets.

The current implementation in MUPAD-COMBINAT does not consider all admissible combinatorial operators as well as restricted cardinalities in sets or powersets. We have added some of these operators in our implementation. In particular, we have considered admissible operators like

$$\varphi(\mathcal{B}, \text{card}\, \tau\, k)$$

where $\varphi \in \{\mathsf{Seq}, \mathsf{Set}, \mathsf{PowerSet}\}$, $\tau \in \{\leq\ /\ =\ /\ \geq\}$ and $k \in \mathbb{N}$.

By the way, the required average CPU time for the implemented operators is clearly competitive. Now one of the following open problems is to develop more efficiently the corresponding pre–computed tables of counting for powersets and sets (see the described isomorphisms for powersets and sets in Section II and the corresponding relation of ordinary generating functions for counting in Table I).

Appendix details the used operators and specifications about our implementation.

IV. CONCLUSIONS AND FUTURE WORK

We have implemented in MUPAD the unranking applied to some basic unlabeled and labeled admissible combinatorial operators: disjoint unions, Cartesian products, and sequences. We are now working on the implementation for (unlabeled) powersets and sets (with and without restricted cardinalities) and cycles.

Our implementation is making two main improvements for the unranking of unlabeled and labeled admissible classes in front of the implementation in the package MUPAD-COMBINAT. First, we have significantly reduced the average CPU time required to generate a random unranking. Second, we are programming more unlabeled admissible combinatorial operators (powersets and sets with and without restricted cardinalities).

Future work is to implement even more unlabeled admissible combinatorial operators (substitution, the open problem for unlabeled cycles, the union among non-disjoint classes, the intersection among classes, etc.).

Another line of research is to study and implement ranking, random generation and exhaustive generation problems described in Section I [34], [19], [23].

On the other hand, it is also interesting to implement similar algorithms for other programming languages as MAPLE, SageMath, C++ or Python, among others.

Moreover, note that these algorithms are also able to develop some structures useful to generate molecules [8], [9] applied to chemistry and influence graphs [24] applied to game theory and social networks, among other topics [29], [36], [7], [13], [33], [1], [5].

Finally, to what we know, it is still open to study the unranking, ranking and exhaustive generation of combinatorial structures using genetic algorithms [22], [32], (meta-)heuristic algorithms [31], [30], [16] or parellel programing [3].

V. APPENDIX

A decomposable combinatorial class can be recursively builded from basic classes using a constructor. The basic classes that we use (and on which other classes are builded) are the following:

- *Epsilon(A)*: a class containing a single object of size 0.
- *Atom(A)*: a class containing a single object of size 1.
- *Z*: equivalent to Atom(Z); standard nomenclature for the definition of Atom.

The avaiable constructors for the building of new classes are:

- *Union(A, B, ...)*: disjoint union of the classes A, B, ...
- *Prod(A, B, ...)*: partitional product of the classes A, B, ...
- *BoxedProd(A, B, ...)*: (only valid for labeled structures) similar to Prod, but forcing the smallest label to be in the leftmost class (or rightmost).
- *Sequence(A)*: all sequences of elements of A, where A should not contain elements of size 0.
- *Set(A) / MultiSet(A)*: all sequences of elements of A with possible repetitions, where A should not contain elements of size 0.
- *PowerSet(A)*: (only valid for unlabeled structures[5]) all sequences of elements of A without repetitions, where A should not contain elements of size 0.

[5]Used in labeled structures, PowerSets behaves as Set(A) / MultiSet(A), because it allows repited elements but with different labeling. So, PowerSet only makes sense used with unlabeled structures.

- *Cycle(A)*: all directed cycles of elements of A without the void cycle, where A should not contain elements of size 0.
- *Alias(A)*: an alias for the class A.

A *combinatorial specification* is a list (or a set or a table) of some productions of the form $A =< rhs >$, where *A* is the name of the class being defined, and $< rhs >$ is an expression that involves elementary classes, constructors and other classes. *A* is known as *non-terminal*.

With this basis, we have been able to generate some solutions and their implementation to the following problems:

- *Standard Specification Generation*: given a class specification, obtain the same specification in the standard *CNF* (*Chomsky Normal Form*).
- *Ranking*: given a combinatorial class A and an object of this class, calulate its rank (according to a specific order)..
- *Unranking*: given a combinatorial class A and a size n, generate the n-th object of the objects of the class A (according to a specific order).

We studied the problem of generating a standard specification via the creation of a specification in the standard *CNF* format, which is a standard for the representation of grammars. A context free grammar is represented in *CNF* if (and only if) all of its production rules are of the form:

- A $\rightarrow$ BC or
- A $\rightarrow \alpha$

where A, B and C are non-terminal symbols[6], and α is a terminal symbol [7]. Either B or C can be the initial symbol.

In our solution, and to be coherent with the previous version in MuPAD implementation, we allow users to introduce any set, list or equation table as a specification. After that, we analyze that specification through an specially created auxiliary table filled with our set of allowed constructors. Any specification using a constructor not present in this table will be rejected. In addition, this table saves additional information that can be useful to determine if a specification is correct or no

Our specification consider three cardinality restrictions: MinLength -minimum cardinality-, Length -exact cardinality- and MaxLength -maximum cardinality-. We also use Union and Product in non-binary expressions. For example, we consider the specifications B = Union(A, B, C) and T = Prod(Z, Z, Z, Z) which are internally transformed into binary expressions for eficency purposes.

At this point we are ready to use our interface to obtain a standard specification. The list of accepted parameters to our function includes:

- *spec*: this parameter contains the specification introduced by the user.
- *label*: this boolean parameter marks the class as labeled (TRUE) or unlabeled (FALSE). Default value is FALSE.
- *boustrophedonic*: this boolean parameter indicates the order for the atom generation. FALSE is used for lexicoraphic order and TRUE for boustrophedonic.

We have two possible outputs from this function:

1) An error, shown as a small text indicating which criteria has been violated by the given specification.
2) A table that contains information about the given specification introduced by the user, the order, and case unlabeled or labeled. Note that the given specification is in standard form, i.e., in *CNF*.

Below you can see some examples of the use of the interface:

1) Specification of a generic atom
   Input
   ```
   ToGenerateStandardSpecification
    ({A = Atom()}, FALSE, FALSE);
   ```
   Output
   ```
   table( {A=Atom()}=table(
   A=Primitive(1,A),
   Z=Primitive(1, Z)),
   MainNonTerminal=A,
   BoustrophedonicOrder=FALSE,
   labeled=FALSE)
   ```
2) Specification of unlabeled binary trees in lexicographic order
   Input
   ```
   ToGenerateStandardSpecification
        ({B=Union(Z,Prod(B,B))}, FALSE,
         FALSE);
   ```
   Output
   ```
   table({B=Union(Z,Prod(B,B))}=table(
   sub1=Prod(B,B),
   B=Union(Z,sub1),
   Z=Primitive(1, Z)),
   MainNonTerminal=B,
   BoustrophedonicOrder=FALSE,
   labeled=FALSE)
   ```

Once we have our standard specification, we can unrank an object of the specified class. To do this, we have implemented an interface which admits the following parameters:

- *spec*: a standard specification.
- *size*: an integer parameter (different from 0), that indicates the size of the objects to generate.
- *rank*: an intger parameter (between 1 and the number of objects of size *size* in the class), which indicates the position in the class of the object to generate.

Some examples of the use of this function are shown below:

1) Generation of the first unlabeled binary tree in lexicographic order of size 4.
   Input
   ```
   bn := ToGenerateStandardSpecification
         ({B=Union(Z,Prod(B,B))}, FALSE,
          FALSE);
   funcioUnrank(bn, 4, 1);
   ```
   Output
   ```
   Prod(Z, Prod(Z, Prod(Z, Z)))
   ```
2) Generation of the third labeled binary tree in lexicographic order of size 3.
   Input

[6]A non-terminal is a symbol that maps a grammar production and can generate strings by replacing both non-terminal and terminal symbols, or a combination of both

[7]A terminal is a symbol that represents a constant value, and therefore can not be decomposed.

```
bn := ToGenerateStandardSpecification
      ({B=Union(Z,Prod(B,B))}, TRUE,
       FALSE);
funcioUnrank(bn, 3, 3);
```

Output

```
Prod(Z(3), Prod(Z(1), Z(2)))
```

REFERENCES

[1] F. Krief A. Amraoui, B. Benmammar. Cognitive radio resource management using multi-agent systems, auctions and game theory. *WSEAS Transactions on Computers*, 13(41):463–475, 2014.

[2] MuPAD-Combinat – open-source algebraic combinatorics package for the computer algebra system MuPAD. URL: http://mupad-combinat.sourceforge.net/.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, The Massachusetts Institute of Technology, 1990.

[4] C. Creutzig and W. Oevel. MuPAD *Tutorial*. SciFace Software (SciFace), Paderborn, 2004.

[5] A. Hulková E. Milkov´ a. Algorithmic and logical thinking development: Base of programming skills. *WSEAS Transactions on Computers*, 12(2):41–51, 2013.

[6] S. Even. *Combinatorial Algorithms*. MacMillan, New York, 1973.

[7] J. Ma F. Wu. The stability, bifurcation and chaos of a duopoly game in the market of complementary products with mixed bundling pricing. *WSEAS Transactions on Mathematics*, 13(35):374–384, 2014.

[8] P. Flajolet and B. Salvy. Computer algebra libraries for combinatorial structures. *J. Symbolic Computation*, 20:653–671, 1995.

[9] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-upsilon-omega: The 1989 cookbook. Technical Report 1073, INRIA, 1989.

[10] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Counting and generating functions. Technical Report 1888, INRIA, 1993.

[11] P. Flajolet and J.S. Vitter. Average-case Analysis of Algorithms and Data Structures. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9. North-Holland, 1990.

[12] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of combinatorial structures. *Theoretical Computer Science*, 132(1):1–35, 1994.

[13] A. Ma J. Ma. Research on the revenue-sharing mechanism based on the price game of retailers. *WSEAS Transactions on Mathematics*, 13(47):484–492, 2014.

[14] D.L. Kreher and D.R. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press LLC, 1999.

[15] Greg Kuperberg, Shachar Lovett, and Ron Peled. Probabilistic existence of regular combinatorial structures. *CoRR*, abs/1302.4295, 2013.

[16] M.A. El-Sharkawi K.Y. Lee. *Modern Heuristic Optimization Techniques: Theory and Applications to Power Systems*. Wiley-IEEE Press.

[17] J. Liebehenschel. Ranking and unranking of lexicographically ordered words: An average-case analysis. *J. of Automata, Languages and Combinatorics*, 2(4):227–268, 1997.

[18] J. Liebehenschel. Ranking and unranking of a generalized dyck language and the application to the generation of random trees. In *The Fifth International Seminar on the Mathematical Analysis of Algorithms*, Bellaterra (Spain), 1999.

[19] A. Lorenz and Y. Ponty. Non-redundant random generation algorithms for weighted context-free languages. *Theoretical Computer Science, Elsevier, 2013, Generation of Combinatorial Structures*, 502:177–194, 2013.

[20] C. Martínez and X. Molinero. A generic approach for the unranking of labeled combinatorial classes. *Random Structures & Algorithms*, 19(3-4):472–497, 2001.

[21] C. Martínez and X. Molinero. Efficient iteration in admissible combinatorial classes. *Theoretical Computer Science*, 346(2–3):388–417, November 2005.

[22] M. Mitchell. *An Introduction to Genetic Algorithms (Complex Adaptive Systems)*. The MIT Press.

[23] X. Molinero. *Ordered Generation of Classes of Combinatorial Structures*. PhD thesis, Universitat Politècnica de Catalunya, November 2005.

[24] X. Molinero, F. Riquelme, and M. J. Serna. Cooperation through social influence. *European Journal of Operation Research*, 242(3):960–974, May 2015.

[25] MuPAD: The computer algebra system. URL: http://es.mathworks.com/discovery/mupad.html.

[26] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms: For Computers and Calculators*. Academic Press, Inc., 1978.

[27] J.M. Pallo. Enumerating, ranking and unranking binary trees. *The Computer Journal*, 29(2):171–175, 1986.

[28] E.M. Reingold, J. Nievergelt, and N.Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[29] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, MA, 1996.

[30] E. Talbi. *Metaheuristics: From Design to Implementation*. Wiley.

[31] P. Vasant. *Meta-Heuristics Optimization Algorithms in Engineering, Business, Economics, and Finance*. IGI Global.

[32] R. Keller W. Banzhaf, P. Nordin and F. Francone. *Genetic Programming An Introduction*. San Francisco, CA: Morgan Kaufmann, 1998.

[33] J. Ma W. Si. Corporation in a closed-loop supply chain based on remanufacturing. *WSEAS Transactions on Mathematics*, 12(4):482–490, 2013.

[34] Y. Wei. The grouping combinaton generating algorithm. In *Proceedings of the International Conference on Computer, Network Security and Communication Engineering (CNSCE 2014)*, pages 670–674, 2014.

[35] H.S. Wilf. East side, west side ... an introduction to combinatorial families- with MAPLE programming. Technical report, 1999. URL: http://www.cis.upenn.edu/˜wilf/lecnotes.html.

[36] J. Ma Y. Yang. Complexity analysis in evolutionary game system in the real estate market. *WSEAS Transactions on Mathematics*, 13(9):79–94, 2014.